Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Deploying Ethereum Infrastructure and Dapps

Final Degree Thesis
Telematic Engineering

Autor: Alberto Miras Gil
Advisor: Jose Luis Muñoz Tapia

Universitat Politècnica de Catalunya (UPC)
2018 - 2019

Abstract

Increasingly nowadays, the word *blockchain* is becoming more common both in the business and in the education field, since in recent years, this technology has revolutionized the world of technology and it has grown in importance exponentially.

In Ethereum-based networks, resources within the network are represented with an identifier of 40 hexadecimal characters length. These identifiers can, for example, represent users or users' *wallets*. In general, it's not easy to memorize these identifiers, which are made up of letters and numbers just for the simple fact of being able to remember your own *wallet*. That is why a system of identification of resources in the Ethereum networks is very important.

In the development of this project, an Ethereum network will be created in collaboration with a set of entities and a DApp (Decentralized Application) will be developed to be able to implement a resource identification system on the previously created network.

Resum

Cada cop més, la paraula *blockchain* s'està tornant més habitual tant en l'àmbit empresarial com en l'àmbit de l'educació, ja que en els últims anys, aquesta tecnologia ha revolucionat el món de la tecnologia i ha crescut en importància exponencialment.

En les xarxes basades en Ethereum, els recursos dintre de la xarxa són representats amb un identificador de 40 caràcters hexadecimals de longitud. Aquests identificadors poden representar per exemple a usuaris o a les *wallets* d'aquests usuaris. En general, no es gens fàcil poder memoritzar aquests identificadors formats per lletres i números només pel simple fet de poder recordar-nos de la nostra propia *wallet*. És per això que un sistema d'identificació de recursos en les xarxes Ethereum és molt important.

En el desenvolupament d'aquest projecte, es crearà una xarxa Ethereum en col·laboració amb un conjunt d'entitats i es desenvoluparà també una DApp (Decentralised Application) per a poder implementar un sistema d'identificació de recursos en la xarxa anteriorment creada.

Resumen

Cada vez más, la palabra *blockchain* se está volviendo más habitual tanto en el ámbito empresarial como en el ámbito de la educación, ya que en los últimos años, esta tecnología ha revolucionado el mundo de la tecnología y ha crecido en importancia exponencialmente.

En las redes basadas en Ethereum, los recursos dentro de la red son representados con un identificador de 40 caracteres hexadecimales de longitud. Estos identificadores pueden representar por ejemplo usuarios o las *wallets* de estos usuarios. En general, no es nada fácil poder memorizar estos identificadores formados por letras y números sólo por el simple hecho de poder acordarte de tu propia *wallet*. Es por ello que un sistema de identificación de recursos en las redes Ethereum es muy importante.

En el desarrollo de este proyecto, se creará una red Ethereum en colaboración con un conjunto de entidades y se desarrollará también una DAPP (Decentralised Application) para poder implementar un sistema de identificación de recursos en la red anteriormente creada.

Acknowledgements

I would like to thank all those who have collaborated in some way in the development of this project.

First of all, I thank the advisor Jose Luis Muñoz Tapia for his dedication and confidence in me, leaving me autonomy and responsibilities when making important decisions of the project. I also thank you for introducing me to the Blockchain world, both in the software development part and in the functional part of this big technology.

I also appreciate the help offered and the guide during the project to Adrià Massanet, a person without whom this project has not been possible, to show interest in my progress and to help me make important decisions during the project.

A special thanks to all those people from the company where I have carried out my curricular practices as well as the University, thanks for your participation and help by giving your opinion, teaching and advising during the process of carrying out this project.

Finally, thank my family and friends for their support and help during this project.

Thanks for everything.

Revision history and approval record

Revision	Date	Purpose
0	31/05/2018	Document creation
1	28/06/2018	Document revision
2	02/07/2018	Document final revision

Document distribution list.

Name	e-mail
Alberto Miras Gil	alberto.miras.gil@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	30/06/2018	**Date**	02/07/2018
Name	Alberto Miras Gil	**Name**	Jose Luis Muñoz Tapia
Position	Autor	**Position**	Supervisor

Contents

List of Figures

Chapter 1

Introduction

1.1 Statement of purpose

This project has been carried out due to the need to have a blockchain network in Barcelona so that students from universities, research groups or any person can create new projects and applications based on this technology. In the national territory, there is already a blockchain network that looks similar to the one created in this project, the *Alastria* [5] network, but it is still growing, it is formed by many private entities and is quite permissioned (unlike the network that we want to create).

The main objectives of this project are:

- The creation of a node that will be part of the network called *NouNetwork* in Barcelona.
- The development of an application that allows the identification of resources within the network previously created.

1.2 Requirements and specification

To carry out this project, the help of Adrià Massanet has been essential, as he was behind the idea of the creation of a new blockchain network in Barcelona, an idea on which this whole project is based.

The node of the network that was created is running with *Geth*, in a machine provided by the project supervisor.
The software development part of this project has been done with the *Solidity* programming language for the SmartContracts. The application tests have been done with the *Truffle* tool and it has been written in *JavaScript*.

1.3 Work Plan

The project will be divided into two main blocks:

- The first block will begin with the documentation and investigation of two large platforms in the blockchain world, Ethereum and Hyperledger Fabric. Once this first part has been done, the creation of a node, the network development and its corresponding documentation will be carried out.

- The second and last block will consist on the development of a DApp, the Ethereum Name Service (ENS). There will also be a part of documentation about the procedure and a test of this DApp.

1.3.1 Gantt Diagram

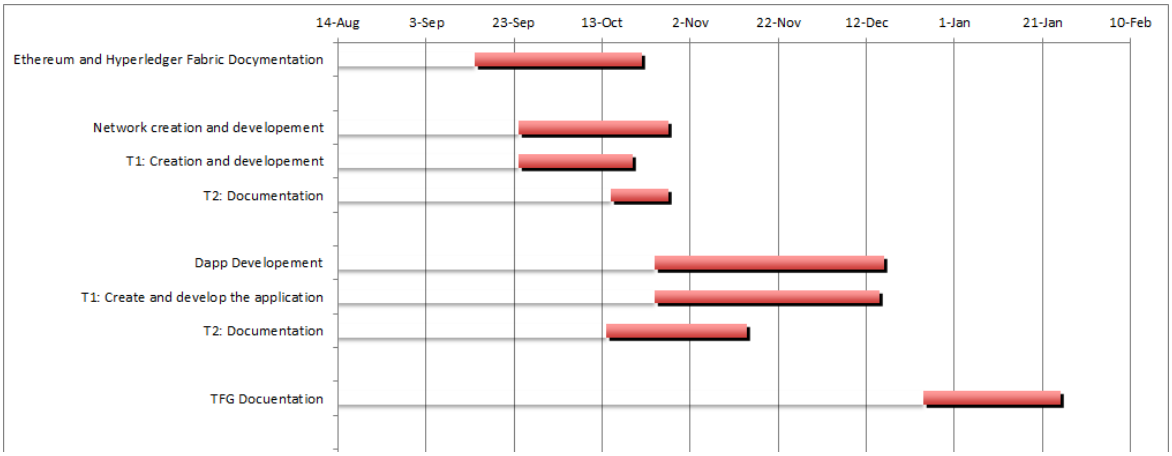

Figure 1.1: Gantt Diagram

Chapter 2

Basic concepts

Within the blockchain technology we find a large set of projects, ecosystems and crypto currencies with different applications and utilities and so many countries begin to create partnerships and alliances to promote this technology. For example, Bitcoin is the most known crypto currency, and Ethereum is one of the biggest projects in blockchain.

Each one of the different projects or platforms that are created under the blockchain technology have a different functionality, so blockchain technology is useful in many different areas. For example, Bitcoin is a decentralized crypto currency and the Ethereum platform is used to create decentralized applications based on blockchain and also has a crypto currency, etc.

2.1 Blockchain

A blockchain is a peer-to-peer decentralized and distributed database where the stored data can't be modified once it is published. It is a public ledger that saves data (transactions) across multiple computers. It is made of blocks that are linked between them using cryptography. Each block of the chain contains the time stamp of its creation, a cryptographic hash of the previous block and the transaction data. Due to the relationship between the blocks, the chain can only be updated with new ones, it can not be modified or any block previously published can not be altered.

The four main caracteristics of blockchain are:

- It's a decentralized technology, there is no need of a centraliced trusted party.
- Cryptography is used to improv the security and the integrity.
- The immutability. Each block has information about its predecesor, so it's almost computationally impossible to modify any previous transaction or block.
- It's a distributed technology as each node of the network has a copy of the ledger.

As a decentralized system, every node has a copy of the ledger or the blockchain so there is not a centralized copy and no user is trusted more than another. Nodes are responsible for validating transactions and creating new blocks with many transactions in it and then they broadcast the block through the network to the other nodes.

The data confirmation is achieved through a consensus process between the participating nodes. The most used algorithm type is the proof of work (PoW) in which there is a competitive process of validation of new entries called mining.

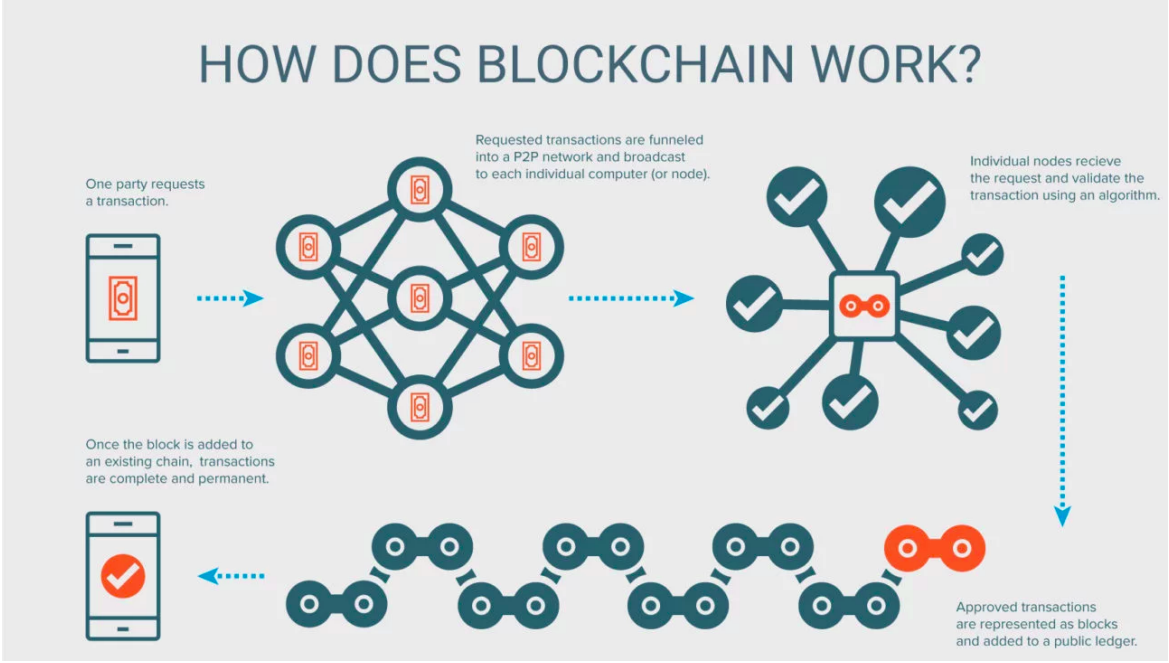

Figure 2.1: How does blockchain work? [1]

There are different types of blockchains [6]:

- Public (permissionless): No access control is required. Applications can be added to the network without any veryfication or approval by other users. Ethereum and Bitcoin are some examples.
- Private (permissioned): This networks use an access control layer to restrict the access. A big difference between the public networks is that validator nodes are supervised or vetted by the network owner. They do not trust in anonymous nodes to validate the network transactions. Hyperledger and Quorum are some examples.

Nowadays, the principal use of a blockchain is a distributed ledger for crypto currencies (for example Bitcoin), but it has many other uses:

- Cryptocurrencies.
- Public administration.
- Financial industry.
- Supply chians monitoring.
- Property registry.
- Carsharing.
- Digital Identity.

2.1.1 Peer to Peer Network (P2P)

The P2P network is a fundamental part of blockchain networks.
In P2P networks, each peer (known as "node") is considered equal among others; each node shares its process power, ability to store data or network bandwith to the other members of the network without the need of third parties.

The nodes called "full node" are responsible for keeping a copy on the device of the entire blockchain while it is connected to the network. This means that to destroy or eliminate the blockchain, all these "full nodes" of the network should be eliminated, and this is impossible.

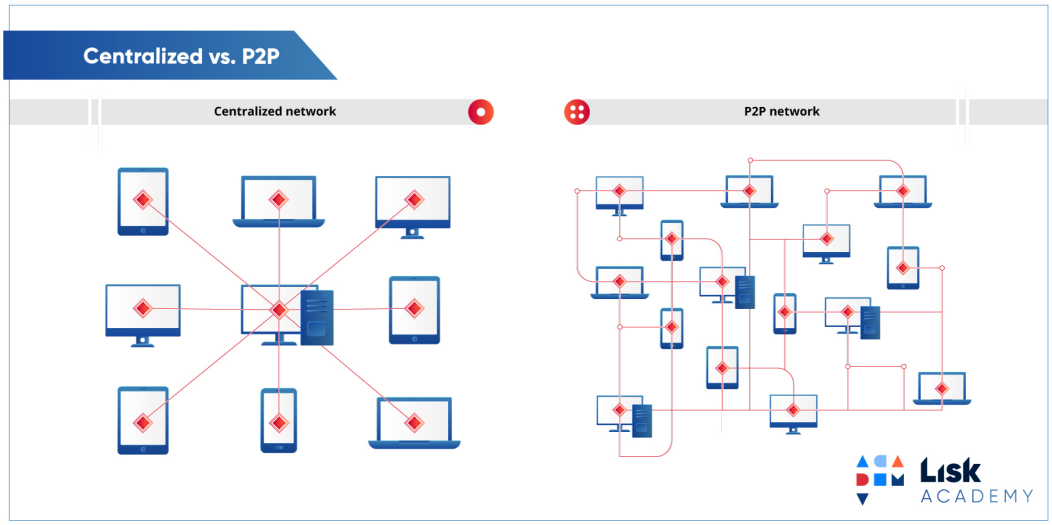

Figure 2.2: Network structure comparison [2]

2.1.2 Consensus protocols

The consensus protocols for blockchain networks are responsible for keeping all nodes in the network synchronized between them.
These protocols allow the network to be updated and ensuring that all the blocks in the chain are correct. Thanks to this protocol, network nodes are allowed to accept data even if one or more nodes fail or are not trusted.

There are different types of consensus protocols. Wwe will explain the most known ones:

- Proof of Work (PoW). To avoid problems and to be able to accept blocks, nodes must perform a set of mathematical algorithms. The first node that is able to overcome these algorithms is rewarded. This consensus protocol requires a great computational capacity.
- Proof of Stake (PoS). The node responsible for creating a new block is chosen deterministically among the wealthiest nodes. The most rewarding nodes are those that will have more chances of creating new blocks.
- Proof of Authority (PoA). It does not require any computational power, since the nodes responsible for creating new blocks are those that have permissions or authority.

Chapter 3

Infrastructure Deployment

In this chapter we will talk about the first step of our project, the creation of a blockchain network called "*NouNetwork*".

Initially, we had to do a preliminary study before the creation of the network, to evaluate and compare two types of blockchain networks, to see what is the most appropriate for our needs. Ethereum [7] was studied as a public and permissionless network, and Hyperledger Fabric [8] was studied as a permissioned network.

3.1 Purpose and objectives

The purpose of the creation of this infrastructure at Barcelona is due to the need for a common environment to be able to develop automated consensus through smartcontracts, which may be a mechanism with a wide implementation in the near future. In the future, this network can be used, for example, to be able to do tests or to deploy smartcontracts or DApps that are part of the final project of students or research groups.
All this project is guided under the XOLN idea of GuifiNet [9]:

- You are free to use the network for any purpose as long as it does not harm the operation of the network itself, the freedom of other users, and respect the conditions of the contents and services.
- You are free to know what the network is, its components, how it works and divulge its spirit and functioning.
- You are free to incorporate services and content to the network with the conditions you want.
- By incorporating yourself to the network, you help to extend these freedoms under the same conditions.

In this infrastructure, there are 8 entities:

- UAB, managed by Jordi Herrera.
- Jose Luis Muñoz, UPC, managed by Jordi Herrera.
- Guifi.net, managed by Agustí.
- White Hat Group, managed by Jordi Baylina.
- Ethereum Dev Barcelona, managed by Adrià Massanet.
- UIB, managed by Macià Mut i Magdalena Payeras.

- UdG, managed by Jordi Herrera, Peplluis de la Rosa.
- URV, managed by Cristina Perez.

As being a decentralized computing network with a limited computing capacity, we will want to achieve that:

- The 50% of the monthly capacity of the network can be assigned directly by the nodes to the projects that are considered opportune.
- The other 50% capacity will be distributed to the projects that request it and that are considered consensual between the nodes.
- The assigned tokens to the projects will be for executing rights in the network and they will not be able to be bought or sold.

The main objectives of this part of the project are:

- Study the platforms of Ethereum and Hyperledger Fabric.
- Choose the platform that best suits our needs.
- Create a network node and connect it to other nodes.
- Install a status server to be able to see information on the network.

3.2 Blockchain technologies

As commented above, an initial study has been carried out with the platforms of Ethereum (public and permissionless) and Hyperledger Fabric (permissioned) to be able to choose the most suitable for our project. First we will come with the explanation of each one of the platforms chosen to do this study, and finally we will justify the technology that we have chosen to continue the project.

3.2.1 Ethereum

Ethereum is a decentralized system created by Vitalik Buterin, Gavin Wood and Joseph Lubin. It is not controlled by anyone and is fully autonomous; that means that every interaction within the system is supported by the users that took part in it so any kind of third-party can't take part. Because of being run from volunteers' computers from every country, the system can not be turned down as it has no central point of failure.

Ethereum is an open source platform that encompasses a large ecosystem formed by a public network where you can develop different decentralized applications (DAPP's) and which also has its own crypto currency, the *ether*.
The ether is a digital asset. However it is not only used as a digital currency, it is also used to pay for the resources used to run an application, acting as 'gas', so for every change in the blockchain

that any user can make, a fee has to be paid. Transaction fees are automatically calculated by the EVM based on how much computing power is necessary to run an action or to execute an application.
The aim of this methodology of paying *ether* in exchange for computing power to send transactions in the blockchain, is that the fee to pay is proportional to the charge that this transaction adds to the network, so that users may create well-formed transactions and malicious users are dissauded from overloading the network.

The applications in Ethereum (DAPP's) are made up of smart contracts, which are abstractions of content implemented in high-level programming languages (such as Solidity, Serpent, LLL and Mutan). Once the code is running on the blockchain, it acts like a self-operating program where there is no possibility of fraud, censorship or third-party interference as they execute the code exactly as programmed.
Once the code is compiled to bytecode, it is executed on the EVM, the virtual machine of Ethereum (environment of execution of the smart contracts).

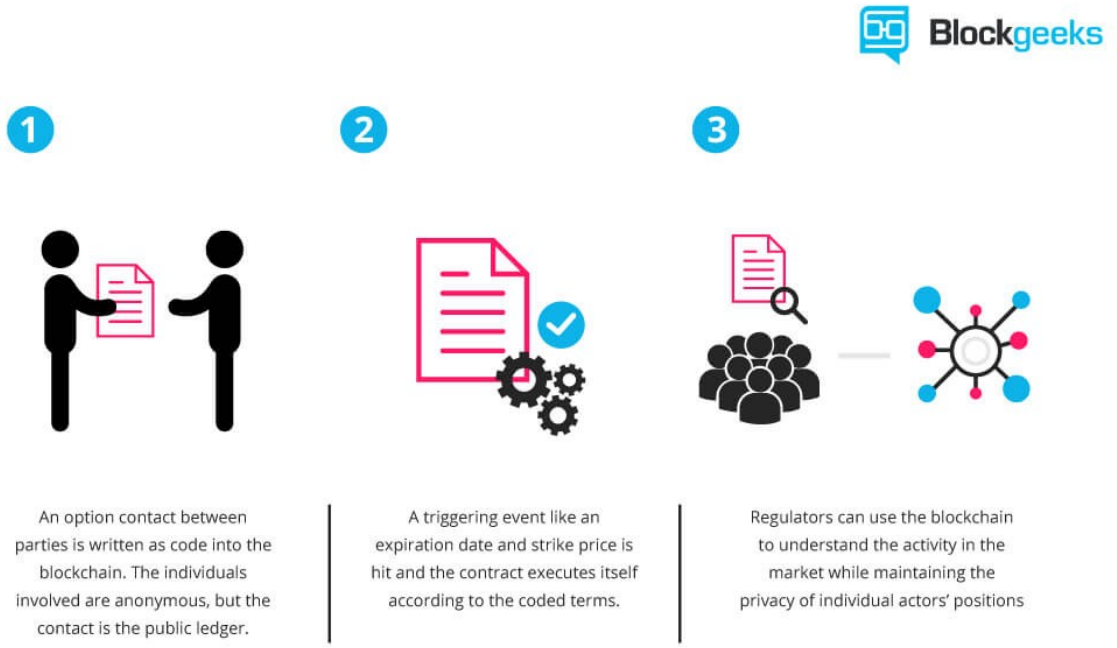

Figure 3.1: Smart Contract utility [3]

In public networks such as Ethereum, there are mainly two mechanisms that ensure the protection of the network:

- All transactions are executed and validated on all the nodes that make up the network.
- There is a consensus algorithm in which all the nodes execute in the same order all the transactions.

In Ethereum, the model to execute the transaction, is the one called "Order-Execute":

1. Creation and propagation of transactions: the injected transactions by the client applications, before being executed, are transported to all nodes in the network.

2. Selection of the node that orders transactions, creates the block and spreads it to the other nodes.
3. The transaction execution and the update of the network status: each node that receives the block, executes all transactions that form it in the same order they apear inside it. Finally, they add the block to the chain.

At present, in Ethereum there is a problem of scalability of the network. This problem occurs because nodes that secure the network, the "full nodes", save a copy of the entire chain. Through the years and the use of the network, this chain has been greatly increasing its size, and it will arrive a moment that it will not be possible or efficient for the nodes to store the entire chain in its own device. For this reason, several solutions to this problem are being considered:

- Plasma chains. This solution is an application that allows you to generate side chains called *child*. These chains share a root, which is the main chain (*mainstream*) of the Ethereum mainnet. These side chains are used to derive the processing of transactions and to make the network more efficient.
- Sharding. The *sharding* consists on dividing the data into different parts that can be stored in different nodes, allowing each node to handle only a small part of the network, thus increasing the performance in the processing of transactions per second.

3.2.1.1 Alastria

An aplication of this technology explained above is *Alastria* [5]. The *Alastria* network is the first permissioned national public network based on blockchain in Spain, which has been designed to help associates to experiment with this blockchain technology in a cooperative environment within the existing regulation [5]. *Alastria* works with *Quorum* [10], which is a fork from Ethereum. *Quorum* is a distributed ledger protocol with transaction privacy and different consensus mechanisms from Ethereum in order to enable this privacy.
This consortium was announced in October 2017, where more than 70 large companies [5] are involved, such as BBVA, CaixaBank, Telefonica, Banco Santander, Universitat de Girona, etc. As it is formed by different entities, Alastria allows its developers to create apps that can be consistent and operable between these institutions.

Alastria will allow [5]:

- The management of digital identity.
- The representation of business assets through tokens, in order to facilitate their transfers and transactions.
- Simplify the business processes that nowadays involve a large number of companies.

3.2.2 Hyperledger Fabric

Hyperledger Fabric is an open source permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts [11]. This platform is contributed by IBM and Digital

Assets and is one of the projects from the Hyperledger Project hosted by the Linux Fundation [8].
Hyperledger Fabric is the first distributed ledger platform where its code or smart contracts (as Hyperledger calls it "chaincode") are writen in general programming languages like Java, Node.js or Go, so the enterprises' developers don't have to learn new programming skills in specific languages as Ethereum developers have.

The Fabric platform is a permissioned one, so as opossed to public networks, the nodes within Hyperledger Fabric are a priori known and completely trusted. This platform supports different types of consensus protocols making it a customized platform that can fit in each particular use case, choosing the most suitable in each situation. Unlike Ethereum (for example), crypto currencies are not necessary in Hyperledger Fabric for its operation, neither to mine nor for the execution of code (in this case, the "chaincode"). By avoiding the use of crypto currencies, the platform can be implemented at a lower cost.

There are three different types of nodes in Hyperledger Fabric:

- Client Nodes. The client nodes are the ones that initiate the transactions and send them to the endorsing peers. The endorsing peers are in charge of the simulation of the transactions.
- Peer Nodes. Keep the ledger data in sync across the network.
  - Endorsing peer: It is a type of peer node and is responsible for simulating the transactions sent by the clients before they are distributed through the nodes of the network and the ledger is updated.
- Orderer Nodes. Responsible for the distribution of the transactions.

In the following figure we can see the data flow through the different types of nodes in the Hyperledger Fabric networks:

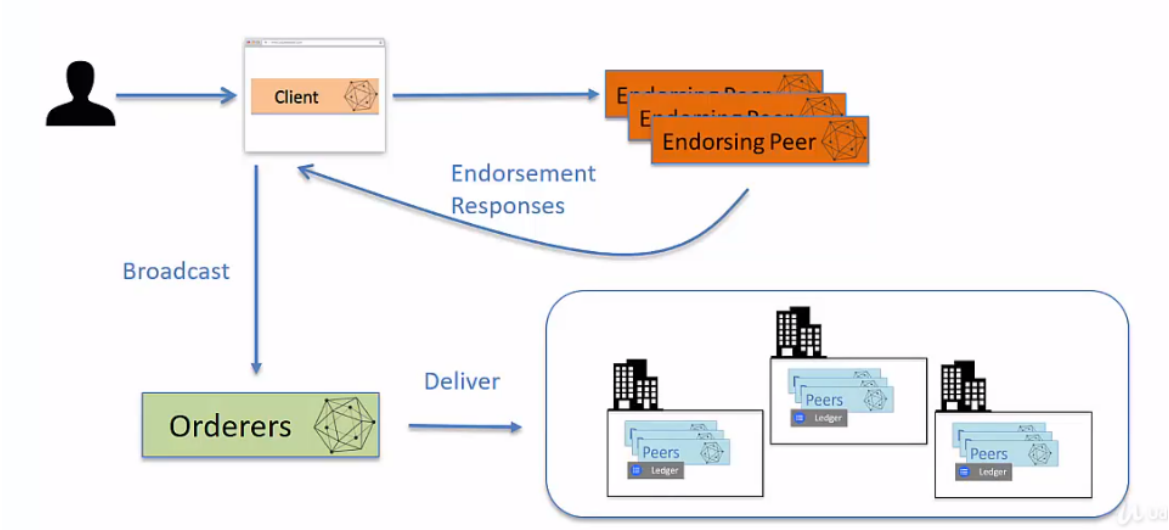

Figure 3.2: Transaction data flow

In Hyperledger Fabric there is the possibility to create channels to privatize the data and transactions sent through them. The members of the this network can participate in more than one channel, due to each transaction in each channel is isolated:

- Peers are connected to the channels.
- Each channel manages its own independent ledger.
- There is no visibility for a peer connected to one channel into the ledger of another channel.

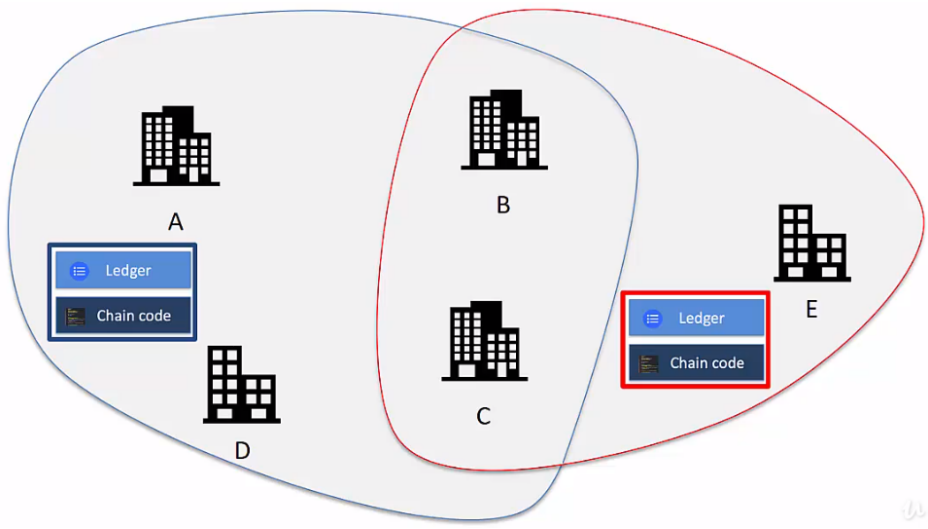

Figure 3.3: Channels in Hyperledger Fabric

The model for the execution of transactions in Hyperledger Fabric is the called "Execute-Order-Validate":

1. Creation and execution of transactions: the transactions that come from the client application, are executed immediately by the set of nodes that received them, although the blockchain status is not updated.
2. The "Ordering Service" orders the client transactions, creates the block and spreads it through the network: this service only takes care of ordering transactions, neither executes nor examines the data.
3. Validate the transactions and update the global state.

3.2.3 Adopted technology

Once we have studied the different platforms and types of blockchain networks, such as public and permissioned, we are already able to choose the most suitable for our project and for our purposes.

Our purposes are similar to Alastria's, a network that facilitates interoperability among the users that make it up. The main differences that can be found between our network and Alastria are that the latter is permissioned and is formed by a large group of companies; while our interest and purpose for *NouNetwork* is to be an available network to any user. We don't want that the entities that make up this network to be private companies with private goals, but to be public entities and share interests.

The network that we want to create has been decided to be public, since paying attention to the aforementioned purposes, it must be accessible to everybody.
Since we want the network to be public and everyone can access it, we have chosen the idea of doing it with Ethereum.
But as long as our network is public, we do not want any user to initialize a node and join the rest of the network, but we want only a few nodes to be in charge of executing the transactions. It has been decided to make it public with permissions, using the *Proof of Authority* (PoA) algorithm [12].

- In the PoA-based networks, transactions and blocks are validated only by a set of approved and known accounts. They are called validators.

To sum up, to be able to develop and implement our blockchain network, we have chosen Ethereum as the most suitable tech, using the PoA algorithm for the consensus between the nodes that will form it.

3.3 Infrastructure Implementation

Here's how we've created an Ethereum node and how the network was created.

First of all you need to know that the node has been done with *Geth*. *Geth* is the command line interface for an Ethereum node (implemented in Go).
Once we have *Geth* installed on our machine, we will create a new ethereum address. This address will be the one of our validator node within the network.

```
1 $ geth account new
```

As discussed above, the transport protocol used within the ethereum networks for communication between nodes is the P2P protocol. Therefore, the next step will be to obtain the public and private key of our P2P node:

```
1 $ bootnode -genkey p2p.key //Private Key of the P2P node
2 $ bootnode -nodekey p2p.key -writeaddress > p2p.address //
    Public key of the P2P node
```

The next step is the *genesis.json* file creation. The genesis block is the first block we find in the chain of a blockchain network. It is similar to a "*settings*" file in our blockchain, where for example you can define the ID of the chain, the difficulty of mining blocks, block time, etc.

In our case we have created the genesis file using the *Puppeth* [13] tool. During the creation of the file, we will have to define a set of parameters:

- The type of consensus to use. In our case we will use Proof of Authority.
- The time between blocks.
- A list of pre-funded address. These are the addresses of each node that forms the network (the validators).

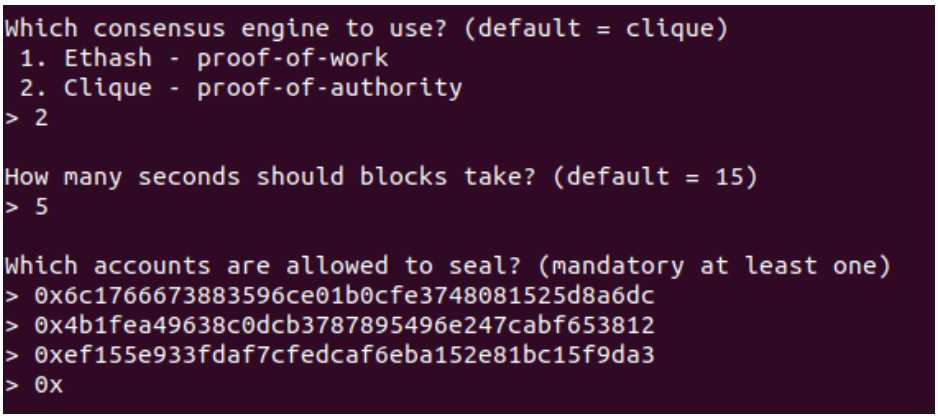

Figure 3.4: Puppeth screen shot

Once the genesis file is created, we just need to attach our node with the other nodes in the network:

```
$    geth --datadir $DATADIR init $GENESISFILE
$    geth \
         --mine \
         --bootnodes $BOOTNODES --datadir $DATADIR \
         --networkid $NETWORKID --nodekey $P2PKEY \
         --keystore $KEYSTORE --unlock $ACCOUNT --password
            $PASSWORD
         --rpc --rpcapi eth,web,personal,clique,networkid \
         --identity $NODEID
```

In this command line we can see that multiple flags of the command *geth* are set [14]:

- *–datadir* Data directory for the databases and keystore.
- *–bootnodes* Comma separated enode URLs for P2P discovery bootstrap.
- *–networkid* Network identifier.
- *–nodekey* P2P node key file.
- *–keystore* Directory for the keystore (default = inside the datadir).
- *–unlock* and *–password* Unlock the given account with the given password.

- *–rpc* To enable the HTTP-RPC server.
- *–rpcapi* API's offered over the HTTP-RPC interface.
- *–identity* Custom node name.

As we can see, in order to execute the code, a set of variables must be defined: the file *genesis.json*, the path to our ethereum keystore, the account's password, the account of our node, the public key of the P2P node and the id of our node.

Once this process is complete, our node is already built up and connected to the blockchain network.

3.3.1 Ethereum Network Status Monitoring

To end the network administration, we have installed a status server to monitor network information at any time. We have used a predefined status server for Ethereum, the Ethereum Network Status [15].In order for the different nodes to connect to the server so their information can be seen and shared, we must add some flags at the time of setting up the nodes, where we define the address and password to access the state server:

```
$    geth --datadir $DATADIR init $GENESISFILE
$    geth --mine --bootnodes $BOOTNODES --datadir $DATADIR \
         --networkid $NETWORKID --nodekey $P2PKEY \
         --keystore $KEYSTORE --unlock $ACCOUNT \
         --password $PASSWORD --rpc \
         --rpcapi eth,web,personal,clique,networkid \
         --identity $NODEID \
         -ethstats $NODEID:$STATSPASSWD@$STATSSERVER
```

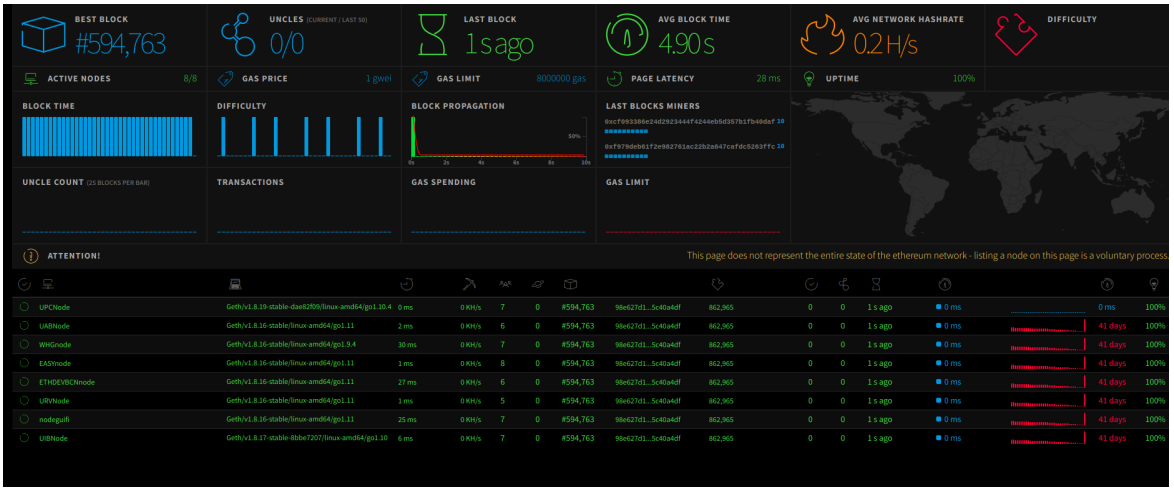

Figure 3.5: Caption of the Ethereum Network Status of our network.

Chapter 4

DApps Development

4.1 Ethereum Name Service (ENS)

Nowadays we find that within Ethereum's blockchain network we work with public addresses to refer to any element or resource inside the network, like smart contracts, users, wallets, etc. These addresses are numbers that have 20 bytes length, and as being represented by hexadecimal numbers, their length is 40 hexadecimal characters (excluding '0x' by which these addresses are preceded). For users of any network or any application, it is difficult to remember a number of 42 digits only to be able to make a transfer from their wallet to a friend. That's why the Ethereum Name Service has been created.

The Ethereum Name Service (ENS) is a system similar to the DNS of the current Internet. Its main objective is to translate these addresses that represent identities or resources in the blockchain, from hexadecimal identifiers to readable names for people, much easier to remember and more simple.

- For example, the address "0xca5AD849706c8E195106774cF584C7490a57bbe0" is translated to "alberto.upc.nn".

The domains in the ENS system are separated hierarchically by dots. The main domains are owned by smart contracts called "registrars", which are responsible for pre-defining a set of rules to be able to manage and to let the users create new subdomains. There are different types of domains at this time:

- '.eth': This extension is related to the mainnet of Ethereum. In order to be able to obtain domains with this extension, bets must be made.
- '.test': This extension is related to Ropsten's test network. The domains ending with this extension are temporary, last for 28 days.

At this moment the system of the ENS is used in different applications and wallets in blockchain networks, like for example [16]: Metamask, Mist, My Ether Wallet, Aragon, Swarm, Etherscan, etc.

4.1.1 Name Resolution

The names or domains within the ENS are managed with a 32-byte hash instead of plain text. This is done for different reasons [4]:

- To mantain the privacy of the names on-chain.

- It simplifies data storage and the domains' processing between smart contracts (they work better with a fixed-length array than with a dynamic length).

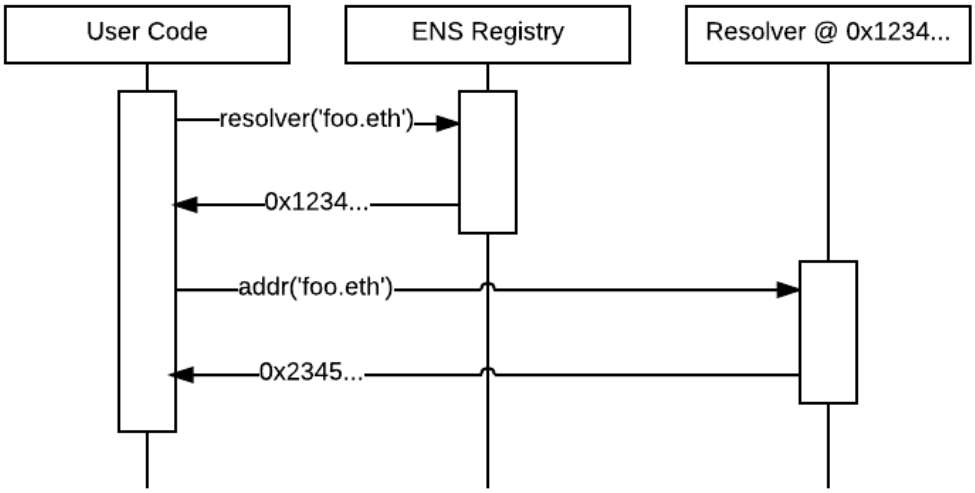

Figure 4.1: Name resolution in ENS [4]

The resolution of names in the ENS follows the procedure that we can see in the figure 4.1:

1. First of all the hash of the name or domain that we want to solve is calculated.
2. Next, we query the registry to find out which resolver is in charge of the domain. The ENS registry will return a '0' if there is no resolver configured for the address we are looking for, or it will reply with the address of the corresponding resolver.
3. Finally we query the resolver with the domain to be solved. The resolver will return the address associated with the desired domain.

4.1.2 ENS Overview

The ENS consists on three main elements, the registry, registrars and resolvers [4].

- **Registrar**: A registrar is any smart contract that owns a domain. It is also responsible for assigning subdomains to users who follow some rules defined in the code.
- **Registry**: It is a unique smart contract that is responsible for storing a list of all domains and subdomains of the system. A part from that, it also stores [4]:
  - The domain owner.
  - The resolver related to the domain.

- **Resolvers**: This element is responsible for translating the Ethereum address into names. A resolver is any smart contract that implements the functionalities defined in the EIP137 standard.

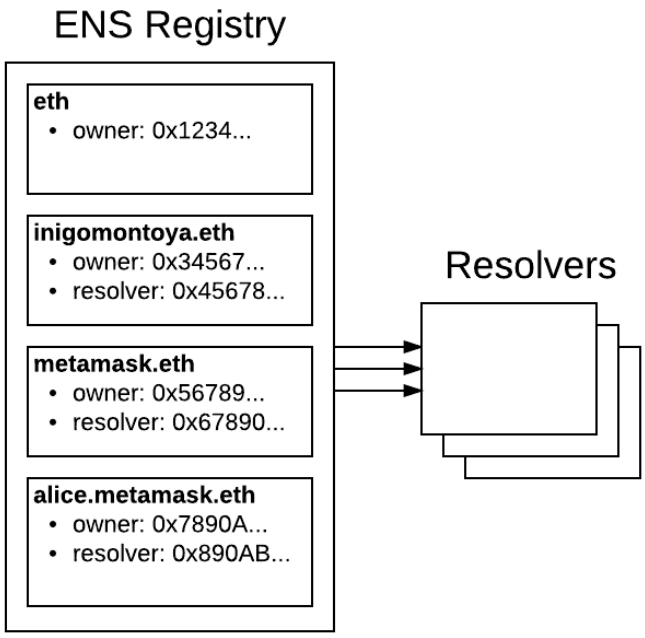

Figure 4.2: ENS Registry overview [4]

4.1.3 Registring a name in the ENS

In the current ENS system we have two ways to register a name, with a FIFS queue (First In Fisrt Served) or by auction.

- Registring a name with the FIFS registrar.
  This type of registration is only used in the Ropsten test network (top-level domain '.test'). As a test network, domains expire 28 days after being registered.
- Registring a name with the auction registrar.
  the Ethereum mainnet. The names that are registered, must have a minimum length of 7 characters (according to the EIP162 standard).

4.1.4 ENS in the Mainnet

The launch of the ENS system on the mainnet ('.eth') began in May 2017 [16] and ended in July of the same year [17]. Once this launch has been completed, we can find a set of curious facts [17]:

- 394,281 auctions started.

- 97,063 auctions finalised
- 11,314 names (12%) configured with resolvers. Here we can appreciate how many users want to use their registered domains.
- 195,259 bids placed.
- 179,503 bids revealed.
- 3,166,895 total ether in bids.
- 161,125 ether locked in names.
- Largest bid: 201,709 ether.
- Largest winning price: 20,103 ether, on 'darkmarket.eth'.
- Most bids: 88 bids, on 'internet.eth'.

4.2 ENS Development

Once the operation of the ENS system has been explained, we will proceed to the explanation of the implementation of this application in our project.

4.2.1 Purpose and objective

Within any network, be it blockchain or not, one of the main objectives during its creation, is the identification of users, entities or resources in it. That is why the Ethereum Name Service (ENS) has been implemented in the new network created previously.

In this project an implementation of the ENS has been made with some modifications (related to the one that is currently running on the mainnet) in the structure and hierarchy of domains. The main idea of these changes is that due to the network consists on 8 nodes which are entities of the Catalan territory, each of these entities can have control and be in charge of managing the users who create subdomains under their domain. As we can see in the figure 4.3:

- There will be a top label domain, the ".nn".
- Each entity will be able to create a first label domain for itself, for example:
  - The UPC will be able to submit the "upc.nn".
  - The UAB will be able to submit the "uab.nn".
- Users that belong to an entity may create subdomains (second label domains):
  - "alberto.upc.nn".
  - "joan.uab.nn".
- In case there are documents or users that need a domain and do not belong to any entity, they can also create a domain. This domain extends from the top level domain ".nn".

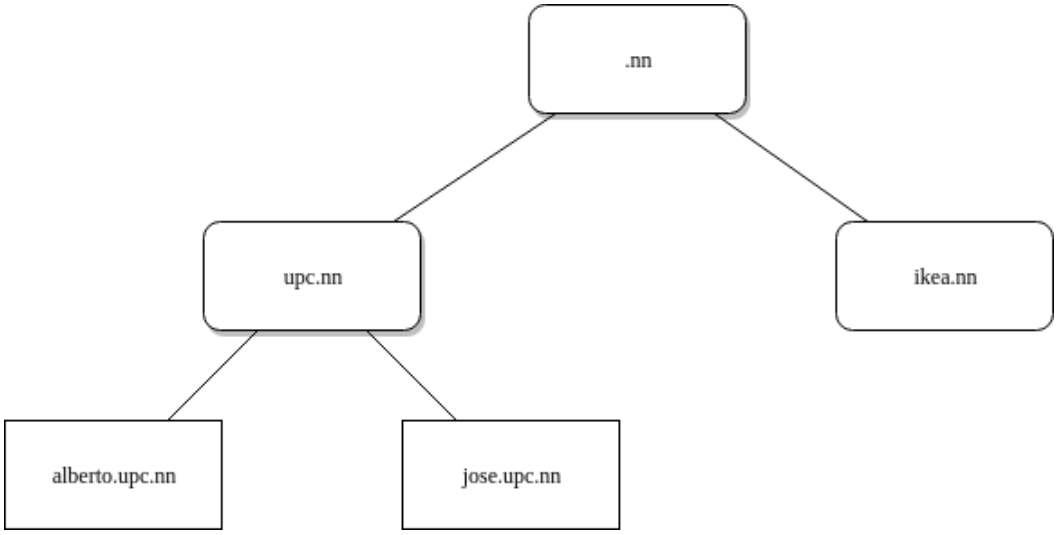

Figure 4.3: Hierarchical names structure

The main objectives of the DApp's implementation are:

- The use of a Multisignature smart contract to manage the transactions by the validator nodes.
- The design of a resource identification system within the blockchain network (the ENS), immutable and unchangeable.
- Creation of the different smart contracts responsible for executing the logic of the ENS.
- Carry out tests of these smart contracts to verify its correct operation.

4.2.2 Multisig

Before beginning with the implementation of the ENS system, we must deal with the Mutlisig smart contract. A Multisig is a smart contract where one or more users must give their approval so that a transaction can be signed and executed.

The network previously created is a public one, so any user can deploy their smart contract and can create any application that they want. At the time of the deployment of critical issues that affect the organization of the system or nodes, it is when the Multisig comes into use.
Before being able to deploy our system of ENS that we have implemented in this project, we will have to submit in the Multisig the transaction that will need to be carried out to deploy our different smart contracts in the network. In order to complete the deployment, a defined number of the nodes in the network must accept.

In summary, the first smart contract to be deployed on the network will be a Multisig, with which the nodes will interact to accept or not relevant transactions within the network.

4.2.2.1 Multisig Implementation

Because the Multisig smart contract must be robust, reliable and secure, the implementation made by Gnosis [18] will be adopted. Gnosis is a prediction market platform which has implemented a smart contract of Multisig that has already passed security audits and is one of the most appropriate and correct for its adaptation within our scope.

In the Multisig smart contract there is a set of main functions:

- Allows to add a new owner. This function has to be called by the Multisig. It ensures that the owner does not exist yet in the Multisig.

```
function addOwner(address owner)
        public
        onlyWallet
        ownerDoesNotExist(owner)
        notNull(owner)
        validRequirement(owners.length + 1, required)
```

- Allows to remove an owner. This function has to be called by the Multisig. It ensures that the owner exists.

```
function removeOwner(address owner)
        public
        onlyWallet
        ownerExists(owner)
```

- Allows to replace an owner with a new owner. This function has to be called by the Multisig.

```
function replaceOwner(address owner, address newOwner)
        public
        onlyWallet
        ownerExists(owner)
        ownerDoesNotExist(newOwner)
```

- Allows to change the number of required confirmations to send a transaction. It ensures that the new number of required confirmations is valid among the number of owners in the Multisig.

```
function changeRequirement(uint _required)
        public
        onlyWallet
        validRequirement(owners.length, _required)
```

- Allows an owner to submit and confirm a transaction. The destination address, the value to send and the data payload of the transaction must be provided.

```
function submitTransaction(address destination, uint value,
    bytes data)
        public
        returns (uint transactionId)
```

- Allows an owner to confirm a transaction.

```
function confirmTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        transactionExists(transactionId)
        notConfirmed(transactionId, msg.sender)
```

- Allows an owner to revoke a confirmation for a transaction.

```
function revokeConfirmation(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
```

- Allows anyone to execute a confirmed transaction if the number of required confirmations is reached.

```
function executeTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
```

4.2.3 ENS Implementation

In the implementation of the ENS we have three main parts:

- **The Registrars**: They will be the two smart contracts in charge of registering the domains and subdomains of the network.
- **ENS Registry**: The registry will have a list of all the domains and subdomains of the network and the resolver associated with each one.
- **Resolver**: It will be the smart contract in charge of translating domains into names.

Each of the four smart contracts that we will explain below have been programmed in Solidity, although it is not the only programming language that exists to code smart contracts.
The code of the four smart contracts is provided in the *Annex*.

4.2.3.1 Domain Registrar

This registrar will be the owner of the top level domain, the ".nn". It will be also in charge of managing the creation of domains by the entities.
This registrar will be able to create two different types of domains:

- The domains related to the entities that are part of the network. Each entity will create its first label domain: "upc.nn", "uab.nn", "guifinet.nn", etc. These new domains belong to a different subdomain registrar for each one.
- Users who are not related to any of the entities that are part of the network can also create their domain.
- Any other type of resource that is within the network, such as documents or smart contracts, may also have an associated domain.

Within this registrar, a control over the users or resources that wish to create a new domain will take place. This means that not any user or resource will be able to create an specified domain; only the addresses related to the nodes that are part of the network can create first label domains of entities (such as "upc.nn").
This control will be carried out thanks to the Multisig smart contract (explained above). Thanks to the fact that all the nodes in the network will have to be part of Multisig before the ENS is created, when submitting any new domain related to an entity, it will be guaranteed that the address is the correct prooving that it is part of the agreement in the Multisig.

The functions that are implemented in this smart contract are:

- With this function, a user can create a new domain from the ".nn" top level domain.
  The domain creation can be controlled or not by the nodes of the Multisig smart contract.
  The option of creating a single domain per user can also be set.
  With this function, the "upc.nn" domain can be created.

```
function createNodeDomain(bytes32 _domain) public
    ifMultisigHasControl(msg.sender)
    onlyOneDomain(msg.sender)
```

- This function creates a new domain from the ".nn" top level domain for a specified address.
  With this function, domains like "ikea.nn" or "bibliotecaBCN.nn" can be created.

```
function createDomain(bytes32 _domain, address contr)
   public
```

- This function enables the control of the new domains created by the nodes that take part in the Multisig smart contract.

```
function multisigControl(bool control) public
    onlyNode(msg.sender)
```

- This function enables or disables the control of one domain per user.

```
function setOnlyOneDomain(bool control) public
    onlyNode(msg.sender)
```

You can find the implementation code in the subsection *6.1.1* of the *Annex*.

4.2.3.2 Subdomain Registrar

The owners of this registrar will be the nodes that make up the network. Each node will only be able to manage and create subdomains under the first label domain of which they are owners; for example, the UPC can only create subdomains that end up in "upc.nn" and will not be able to manage subdomains that end in "uab.nn".

At the moment anyone wants to create a subdomain related to an entity that is part of the network, the user just have to go to the appropriate Subdomain Registrar.

This smart contract implements two principal functions:

- This function creates a new subdomain for a user (who is calling the function). The only parameter that this function needs is the hash of the subdomain to create.
  With this function we can create for example the "alberto.upc.nn" subdomain.

```
function createUserSubDomain(bytes32 _subDomain) public
```

- This function creates a new subdomain for a specified address. The two parameters that this function needs are the hash of the new subdomain to create and the specified address to relate this subdomain with.

```
function createSubDomain(bytes32 _subDomain, address contr)
    public
```

You can find the implementation code in the subsection *6.1.2* of the *Annex*.

4.2.3.3 ENS Registry

This smart contract will be responsible for storing a list of all domains and subdomains, the owners of these domains and the resolvers related to each one of them.
The calls and requests of the two Registrars to the ENS Registry to update or add a new subdomain to the registry list are controlled. That means:

- Only the *Domain Register* can create new domains under the ".nn" top level domain, as this registrar will be the domain owner. For example, this registrar will be the only one that can create domains like "upc.nn".

- Only the *Subdomain Register* of each entity can create new subdomains under its first label domain. For example, the UPC registrar will be the only one that can create domains like "alberto.upc.nn".

To sum up, only registrars, (smart contracts that own a domain) can create new subdomains under its principal domain.

The functions implemented in this smart contract are:

- This function returns the owner of the specified domain.

```
function owner(bytes32 domain) public view returns (address
    )
```

- This function returns the resolver for the specified domain.

```
function resolver(bytes32 domain) public view returns (
    address)
```

- This function transfers the ownership of a domain to a new address. It can only be called by the current owner of the domain. The parameters that this function needs are the domain to transfer ownership of and the address of the new owner.

```
function setOwner(bytes32 domain, address _owner) public
    onlyOwner(domain)
```

- This function creates a new domain for a new user. It can only be called by the current owner of the root domain. Domains can only be created if they don't exist yet.
  For example, this function can only be called by the registrar that owns the ".nn" domain to create a new domain (the "upc.nn" domain for example).

```
function setDomainOwner(bytes32 rootdomain, bytes32 _domain
    , address _owner) public
    onlyOwner(rootdomain)
    onlyIfDoesNotExist(rootdomain, _domain)
    onlyIfDoesNotHaveDomain(_owner)
```

- This function creates a new subdomain for a new user.It can only be called by the current owner of the domain. Subdomains can only be created if they don't exist yet.
  For example, this function can only be called by the registrar that owns the "upc.nn" domain to create a new subdomain.

```
function setSubDomainOwner(bytes32 domain, bytes32
    _subdomain, address subdomainowner) public
    onlyOwner(domain)
    onlyIfDoesNotExist(domain, _subdomain)
    onlyIfDoesNotHaveDomain(subdomainowner)
```

- This function sets the resolver address for the specified domain. The resolver can only be set by the current owner of the domain.

```
function setResolver(bytes32 rootdomain, bytes32 _domain,
    address _resolver) public
    onlyOwner(rootdomain)
```

You can find the implementation code in the subsection *6.1.3* of the *Annex*.

4.2.3.4 Resolver

This smart contract will be in charge of completing the final domain or subdomain translation to the address related to the desired resource.

The functions that are implemented in the Resolver's smart contract are:

- This function is used to translate a domain into its related address.
  The parameter needed for this function is the hash of the queried domain. It returns the address associated with an ENS domain.

```
function addr(bytes32 domain) public view returns (address)
```

- This function sets the address associated with an ENS domain or subdomain. It can only be called by the registrar that owns the root domain.
  The parameters that this function needs are the domain to update, the root to which the domain to update is related to and the address to set.

```
function setAddr(bytes32 root, bytes32 _domain, address
    _addr) public
    onlyOwner(root)
    onlyNotNull(_addr)
```

You can find the implementation code in the subsection *6.1.4* of the *Annex*.

4.2.4 User stories

In this subsection, we are going to explain the work flow of the ENS system that we have developed in different scenarios as a solution to a few user stories.

- As a entity that take part in the Multisig smart contract, I want to be able to create my own domain from which I will be able to create and manage new subdomains. In this case, the entity may interact with the Registrar Domain, the smart contract that owns the ".nn" domain.

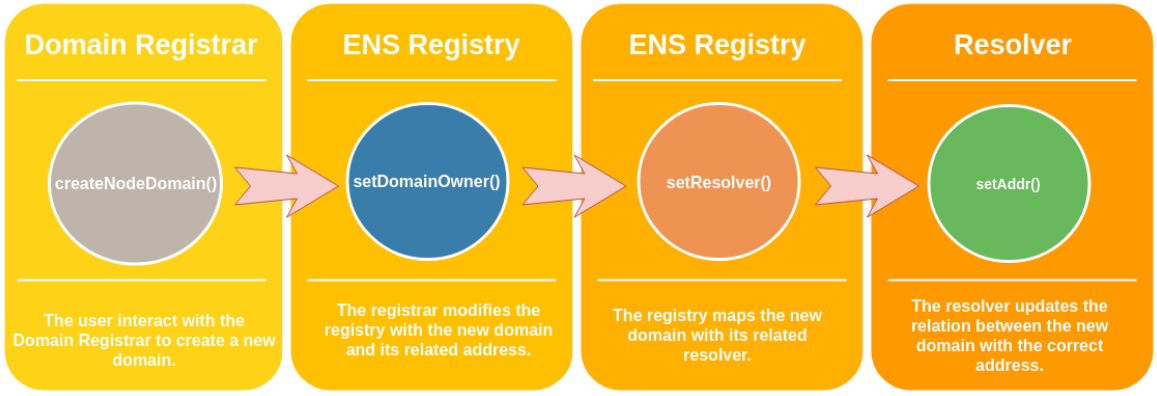

Figure 4.4: Domain creation flow

- As an employee of an entity that is part of the *NouNetwork*, I want to create my own subdomain in the network. In this case, the user interacts with the SubDomain Registrar (that owns the "upc.nn" domain) to create the new desired subdomain, for example the "alberto.upc.nn" subdomain.

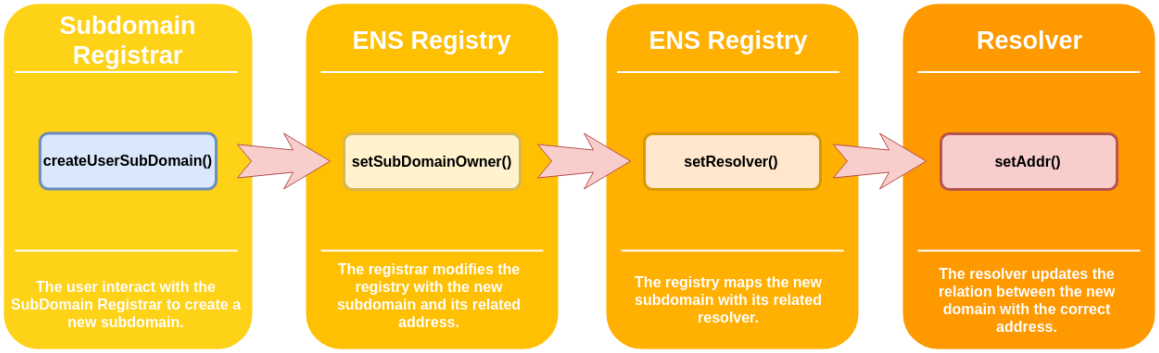

Figure 4.5: Subdomain creation flow

- As a user that is not related to any entity, I want to create my own domain in the network. This case has the same data flow as the one shown in the figure 4.4. The user may interact with the Registrar Domain that owns the ".nn".

4.2.5 DApp Testing

Once the implementation and development of the ENS has been completed, we will proceed to test its correct functioning.
These tests have been made to simulate the correct functioning of the smart contracts within the network. Since the network is still in a first phase of implementation, it has not yet been possible to deploy the system, and therefore, these tests have been carried out in equivalence.

The tests have been developed in *Truffle*. *Truffle* is a development environment and testing framework for Ethereum. Thanks to this tool, we can simulate a blockchain network and create an optimal environment for the test of smart contracts, which is the equivalent to its functionality

in public networks. The tests have been programmed in *JavaScript*; the code can be seen in the subsection *6.2* of the *Annex*.

In the image 4.6 we can see a screenshot of how the above-mentioned tests are passed successfully, therefore, we can ensure and confirm that the developed ENS system is working correctly.

```
truffle(develop)> test
Using network 'develop'.



  Contract: RegistrarDomain & RegistrarSubDomain
    ✓ should have nodes (RegistrarDomain) (488ms)
    ✓ should have root (RegistrarDomain) (338ms)
    ✓ should have set the domain owner and the resolver in the ENS (RegistrarDomain) (305ms)
    ✓ should have Resolver (RegistrarDomain) (346ms)
    ✓ should set a new domain (RegistrarDomain) (687ms)
    ✓ should NOT set a new domain (RegistrarDomain) (sender is not an owner) (515ms)
    ✓ should NOT set a new domain (RegistrarDomain) (domain already exists) (627ms)
    ✓ should NOT set a new domain (RegistrarDomain) (owner has already have a domain) (538ms)
    ✓ should have set a new subdomain (RegistrarSubDomain) (715ms)

  Contract: ENSRegistry
    ✓ should have a root owner (120ms)
    ✓ should set a new domain owner (163ms)
    ✓ should NOT set a new domain owner (sender is not the owner of the root) (142ms)
    ✓ should NOT set a new domain owner (domain already exists) (195ms)
    ✓ should NOT set a new domain owner (owner has already have a domain) (192ms)
    ✓ should set a new sub domain owner (204ms)
    ✓ should NOT set a new sub domain owner (sender is not the owner of the domain) (174ms)
    ✓ should NOT set a new sub domain owner (subdomain already exists) (309ms)
    ✓ should NOT set a new sub domain owner (owner has already have a subdomain) (237ms)
    ✓ should set a new resolver (218ms)
    ✓ should NOT set a new resolver (sender is not the owner of the domain) (191ms)
    ✓ should change the domain owner (217ms)
    ✓ should NOT change the domain owner (sender is not the owner of the domain) (186ms)

  Contract: Resolver
    ✓ should have ENS (114ms)
    ✓ should support the address interface (93ms)
    ✓ should NOT map a domain to an address (sender is not the owner of the domain) (110ms)
    ✓ should NOT map a domain to an address (address is null) (132ms)


  26 passing (8s)

truffle(develop)>
```

Figure 4.6: Truffle Tests

Chapter 5

Conclusions and future development

In a few months, the word *Blockchain* has been becoming more and more important in the business world and technology. At present there are already a large number of companies that work under this big technology to design global networks that allow the distribution of information in a decentralized way. A good example of a use case of this technology is the Alastria network in Spain, which some of its purposes are to solve problems of traceability, fraud and information sharing among clients.

Thanks to discharging the expectations and objectives set at the beginning of this project, I have been able to see how the blockchain's great world is both in the part of the infrastructure as in the programming part.
Firstly, thanks to the first part of the project where I had to create a new node to be able to make a decentralized network in Barcelona, it has allowed me to investigate the different platforms and technologies that currently exist in this technology and I have been able to increase exponentially my knowledge and my vision on the Blockchain technology.
Secondly, thanks to the second part of the project where an identification of users and resources had to be implemented within the network previously created, it has allowed me to improve my knowledge about the programming of SmartContracts and the logic that lies behind this great technology.

Finally, after having done this DApp as a second part of the project, I can suggest some improvements or future steps to follow in the future. As the created network is in an initial development and testing part, the application created has not been deployed and only tests have been performed to ensure its correct operation.
One of the first steps to follow would be to deploy the DApp within the network so that it can be put into operation. Then, once the network has started and users begin to use it, it would be necessary to have a user interface so that they can interact with the application.
The last step that could be done would be the adaptation of this system of identification of resources within the other DApps that are created or are going to be implemented in this network, so that the applications work above this system and they must interact with it.

Chapter 6

Annex

6.1 DApp's SmartContracts

6.1.1 RegistrarDomain.sol

```
pragma solidity ^0.4.24;

import "./ENSRegistry.sol";
import "./Resolver.sol";
import "./RegistrarSubDomain.sol";
import "./MultiSigWallet.sol";

contract RegistrarDomain {

    ENSRegistry public ens;
    Resolver public resolver;
    RegistrarSubDomain public registrarSubDomain;
    MultiSigWallet public multisig;

    bytes32 public root;
    bytes32 public domain;
    mapping(address=>bool) public hasDomain;
    bool public multisigEnabled;
    bool public oneDomain; //Enable or disable the control of
       one domain per user

    event DomainCreation(bytes32 indexed subdomain, address
       indexed owner);

    /*
     * Modifiers
     */

    modifier onlyNode(address node) {
        require(multisig.isOwner(node));
        _;
    }

    modifier ifMultisigHasControl(address node) {
        if(multisigEnabled) {
            require(multisig.isOwner(node));
            _;
        } else {
```

```
37              _;
38          }
39      }
40
41      modifier onlyOneDomain(address node) {
42          if(oneDomain) {
43              require(!hasDomain[node]);
44              _;
45          } else {
46              _;
47          }
48      }
49
50      /*
51       * Getters
52       */
53
54      function getRoot() public view returns (bytes32) {
55          return root;
56      }
57
58      function getDomain() public view returns (bytes32) {
59          return domain;
60      }
61
62      function isOwner(address _owner) public view returns (bool)
            {
63          return multisig.isOwner(_owner);
64      }
65
66      function getENS() public view returns (address) {
67          return ens;
68      }
69
70      function getResolver() public view returns (address) {
71          return resolver;
72      }
73
74      function getRegistrarSubDomain() public view returns (
            address) {
75          return registrarSubDomain;
76      }
77
78      /*
79       * Functions
80       */
81
82      constructor(bytes32 _domain, address _multisig) public {
83          root = _domain;
84
```

```
85          ens = new ENSRegistry(this, root);
86          resolver = new Resolver(ens);
87          multisig = MultiSigWallet(_multisig);
88
89          multisigEnabled = false;
90      }
91
92      // Enables the control of the multisig's owners.
93      function multisigControl(bool control) public
94      onlyNode(msg.sender)
95      {
96          multisigEnabled = control;
97      }
98
99      // Enables or disables the control of one domain per user.
100     function setOnlyOneDomain(bool control) public
101     onlyNode(msg.sender)
102     {
103         oneDomain = control;
104     }
105
106     // Creates a new domain from the principal domain for a
            user.
107     // @param domain -> The domain to create.
108     // @return addr -> The address of the new registrar created
            .
109     function createNodeDomain(bytes32 _domain) public
110     ifMultisigHasControl(msg.sender)
111     onlyOneDomain(msg.sender)
112     {
113         hasDomain[msg.sender] = true;
114         domain = sha3(root, _domain);
115         registrarSubDomain = new RegistrarSubDomain(domain, ens
                , resolver);
116         ens.setDomainOwner(root, _domain, registrarSubDomain);
117         ens.setResolver(root, _domain, resolver);
118         resolver.setAddr(root, _domain, registrarSubDomain);
119
120         emit DomainCreation(domain, registrarSubDomain);
121     }
122
123     // Creates a new domain from the principal domain for an
            especified address.
124     // @param domain -> The domain to create.
125     // @param contr -> The adddress to relate the domain to.
126     // @return addr -> The address of the new registrar created
            .
127     function createDomain(bytes32 _domain, address contr)
            public
128     {
```

```
129         hasDomain[msg.sender] = true;
130         domain = sha3(root, _domain);
131         ens.setDomainOwner(root, _domain, contr);
132         ens.setResolver(root, _domain, resolver);
133         resolver.setAddr(root, _domain, contr);
134 
135         emit DomainCreation(domain, contr);
136     }
137 }
```

6.1.2 RegistrarSubDomain.sol

```
1  pragma solidity ^0.4.24;
2 
3  import "./ENSRegistry.sol";
4  import "./Resolver.sol";
5  import "./RegistrarSubDomain.sol";
6 
7  contract RegistrarSubDomain {
8 
9      ENSRegistry public ens;
10     Resolver public resolver;
11     bytes32 public domain;
12 
13     event SubDomainCreation(bytes32 indexed subdomain, address
           indexed owner);
14 
15     /*
16      * Functions
17      */
18 
19     constructor(bytes32 _domain, address _ens, address
           _resolver) public {
20         domain = _domain;
21         ens = ENSRegistry(_ens);
22         resolver = Resolver(_resolver);
23     }
24 
25     // Creates a new subdomain for a user.
26     // @param _subDomain -> The subdomain to create.
27     function createUserSubDomain(bytes32 _subDomain) public {
28         ens.setSubDomainOwner(domain, _subDomain,  msg.sender);
29         ens.setResolver(domain, _subDomain, resolver);
30         resolver.setAddr(domain, _subDomain, msg.sender);
31 
32         bytes32 subDomain = sha3(domain, _subDomain);
33         emit SubDomainCreation(subDomain, msg.sender);
34     }
35 
36     // Creates a new subdomain for an especified address.
```

```
37      // @param _subDomain -> The subdomain to create.
38      function createSubDomain(bytes32 _subDomain, address contr)
            public {
39          ens.setSubDomainOwner(domain, _subDomain,  contr);
40          ens.setResolver(domain, _subDomain, resolver);
41          resolver.setAddr(domain, _subDomain, contr);
42
43          bytes32 subdomain = sha3(domain, _subDomain);
44          emit SubDomainCreation(subdomain, contr);
45      }
46  }
```

6.1.3 ENSRegistry.sol

```
1   pragma solidity ^0.4.24;
2
3
4   contract ENSRegistry {
5
6       struct Record {
7           address owner;
8           address resolver;
9       }
10
11      mapping(bytes32=>Record) public records;
12      mapping(address=>bool) public addressHasDomain;
13
14      event OwnerChanged(bytes32 indexed domain, address indexed
            newOwner);
15      event NewDomainOwner(bytes32 indexed newDomain, address
            indexed owner);
16      event NewSubDomainOwner(bytes32 indexed newSubDomain,
            address indexed owner);
17      event NewResolver(bytes32 indexed domain, address indexed
            resolver);
18
19      /*
20       * Modifiers
21       */
22
23      // Permits modifications only by the owner of the specified
            domain.
24      modifier onlyOwner(bytes32 domain) {
25          require(records[domain].owner == msg.sender);
26          _;
27      }
28
29      modifier onlyIfDoesNotExist(bytes32 domain, bytes32
            _subdomain) {
30          bytes32 subdomain = sha3(domain, _subdomain);
```

```
31          require((records[subdomain].owner == address(0)) && (
                records[subdomain].resolver == address(0)));
32          _;
33      }
34
35      modifier onlyIfDoesNotHaveDomain(address addr) {
36          require(!addressHasDomain[addr]);
37          _;
38      }
39
40      /*
41       * Functions
42       */
43
44      // Constructs a new ENS Registry, with the provided address
            as the owner of the root domain.
45      constructor(address owner, bytes32 rootdomain) public {
46          records[rootdomain].owner = owner;
47      }
48
49      // Returns the owner of the specified domain.
50      function owner(bytes32 domain) public view returns (address
            ) {
51          return records[domain].owner;
52      }
53
54      // Returns the resolver for the specified domain.
55      function resolver(bytes32 domain) public view returns (
            address) {
56          return records[domain].resolver;
57      }
58
59      // Transfers ownership of a domain to a new address. It can
            only be called by the current owner of the domain.
60      // @param domain -> The domain to transfer ownership of.
61      // @param owner -> The address of the new owner.
62      function setOwner(bytes32 domain, address _owner) public
63      onlyOwner(domain)
64      {
65          records[domain].owner = _owner;
66
67          emit OwnerChanged(domain, _owner);
68      }
69
70      // Creates a new domain. It can only be called by the
            current owner of the root domain.
71      // @param rootdomain -> The root domain.
72      // @param domain -> The new domain to create.
73      // @param owner -> The address of the new owner.
74      function setDomainOwner(bytes32 rootdomain, bytes32 _domain
```

```
        , address _owner) public
75      onlyOwner(rootdomain)
76      onlyIfDoesNotExist(rootdomain, _domain)
77      onlyIfDoesNotHaveDomain(_owner)
78      {
79          bytes32 domain = sha3(rootdomain, _domain);
80          records[domain].owner = _owner;
81          addressHasDomain[_owner] = true;
82
83          emit NewDomainOwner(domain, _owner);
84      }
85
86      // Transfers ownership of a subdomain 'sha3(domain,
           subdomain)' to a new address.
87      // It can only be called by the owner of the domain.
88      // @param domain -> The parent domain.
89      // @param subdomain -> The hash of the subdomain.
90      // @param owner -> The address of the new subdomain.
91      function setSubDomainOwner(bytes32 domain, bytes32
           _subdomain, address subdomainowner) public
92      onlyOwner(domain)
93      onlyIfDoesNotExist(domain, _subdomain)
94      onlyIfDoesNotHaveDomain(subdomainowner)
95      {
96          bytes32 subdomain = keccak256(abi.encodePacked(domain,
               _subdomain));
97          records[subdomain].owner = subdomainowner;
98          addressHasDomain[subdomainowner] = true;
99
100         emit NewSubDomainOwner(subdomain, subdomainowner);
101     }
102
103     // Sets the resolver address for the specified domain.
104     // @param domain -> The domain to update.
105     // @param resolver -> The address of the resolver.
106     function setResolver(bytes32 rootdomain, bytes32 _domain,
           address _resolver) public
107     onlyOwner(rootdomain)
108     {
109         bytes32 domain = sha3(rootdomain, _domain);
110         records[domain].resolver = _resolver;
111
112         emit NewResolver(domain, _resolver);
113     }
114 }
```

6.1.4 Resolver.sol

```
1   pragma solidity ^0.4.24;
2
```

```
3  import "./ENSRegistry.sol";
4
5  contract Resolver {
6
7      ENSRegistry ens;
8      mapping(bytes32=>address) addresses;
9
10     event NewAddress(bytes32 indexed newDomain, address indexed
           user);
11
12     /*
13      * Modifiers
14      */
15
16     modifier onlyOwner(bytes32 domain) {
17         require(ens.owner(domain) == msg.sender);
18         _;
19     }
20
21     modifier onlyNotNull(address _addr){
22         require(_addr != address(0));
23         _;
24     }
25
26     /*
27      * Getters
28      */
29
30     function getENS() public view returns(address){
31         return ens;
32     }
33
34     /*
35      * Functions
36      */
37
38     constructor(address ensAddr) public {
39         ens = ENSRegistry(ensAddr);
40     }
41
42     // Returns the address associated with an ENS domain.
43     // @param domain -> The ENS domain to query.
44     // @return -> The associated address.
45     function addr(bytes32 domain) public view returns (address)
           {
46         address ret = addresses[domain];
47         require(ret != address(0));
48         return ret;
49     }
50
```

```
51      // Sets the address associated with an ENS domain. It can
            only be called by the owner of that domain in the ENS
            registry.
52      // @param domain -> The domain to update.
53      // @param addr -> The address to set.
54      function setAddr(bytes32 root, bytes32 _domain, address
            _addr) public
55      onlyOwner(root)
56      onlyNotNull(_addr)
57      {
58          bytes32 domain = sha3(root, _domain);
59          addresses[domain] = _addr;
60
61          emit NewAddress(domain, _addr);
62      }
63
64      // This function shows us what kind of interface does our
            resolver support.
65      // @param interfaceID -> interface to be checked.
66      // The '0x3b3b57de' ID defines the address interface.
67      // The '0x01ffc9a7' ID defines the method itself.
68      function supportsInterface(bytes4 interfaceID) public view
            returns (bool) {
69          return interfaceID == 0x3b3b57de || interfaceID == 0
                x01ffc9a7;
70      }
71  }
```

6.2 DApps Test

```
1  let RegistrarDomain = artifacts.require("../contracts/
       RegistrarDomain.sol");
2  let ENSRegistry = artifacts.require("../contracts/ENSRegistry.
       sol");
3  let RegistrarSubDomain = artifacts.require("../contracts/
       RegistrarSubDomain.sol");
4  let Resolver = artifacts.require("../contracts/Resolver.sol");
5  let MultiSigWallet = artifacts.require("../contracts/
       MultiSigWallet.sol");
6
7  const {getWeb3, getContractInstance} = require('./helper');
8  const web3_1_0 = getWeb3();
9  const getInstance = getContractInstance(web3_1_0);
10
11 const should = require('chai').use(require('chai-as-promised'))
       .should();
12 const ERROR_MSG = 'VM Exception while processing transaction:
       revert';
13
```

```
14 // ABI's declaration
15 const ENSABI = ENSRegistry.abi;
16 const ResolverABI = Resolver.abi;
17 const RegistrarSubDomainABI = RegistrarSubDomain.abi;
18
19 const registrarDomainSettings = {
20     root: web3_1_0.utils.keccak256('.nounetwork')
21 }
22
23 const multisigWalletSettings = {
24     owners: [web3.eth.accounts[0], web3.eth.accounts[1]],
25     required: 1
26 }
27
28 const ensSettings = {
29     owner: web3.eth.accounts[0],
30     root: web3_1_0.utils.keccak256('.nounetwork')
31 }
32
33
34 contract('RegistrarDomain & RegistrarSubDomain', function(
       accounts){
35
36     it("should have nodes (RegistrarDomain)", async function(){
37         let instance = getInstance('MultiSigWallet', accounts
               [0]);
38         const MultisigInstance = await instance
39             .deploy({ arguments: [multisigWalletSettings.owners
                   , multisigWalletSettings.required]}).send({from:
                    accounts[0]});
40
41         instance = getInstance('RegistrarDomain', accounts[0]);
42         const RegistrarDomainInstance = await instance
43             .deploy({ arguments: [registrarDomainSettings.root,
                    MultisigInstance._address]}).send({from:
                   accounts[0]});
44         let node = await RegistrarDomainInstance.methods.
               isOwner(accounts[0]).call();
45         assert.equal(node, true, "It does not have nodes");
46
47         node = await RegistrarDomainInstance.methods.isOwner(
               accounts[1]).call();
48         assert.equal(node, true, "It does not have nodes");
49     });
50
51     it("should have root (RegistrarDomain)", async function(){
52         let instance = getInstance('MultiSigWallet', accounts
               [0]);
53         const MultisigInstance = await instance
54             .deploy({ arguments: [multisigWalletSettings.owners
```

```
                , multisigWalletSettings.required]}).send({from:
                 accounts[0]});
55
56          instance = getInstance('RegistrarDomain', accounts[0]);
57          const RegistrarDomainInstance = await instance
58              .deploy({ arguments: [registrarDomainSettings.root,
                    MultisigInstance._address]}).send({from:
                   accounts[0]});
59
60          let root = await RegistrarDomainInstance.methods.
               getRoot().call();
61
62          assert.equal(root, web3_1_0.utils.keccak256('.
               nounetwork'), "It does not have root");
63      });
64
65      it("should have set the domain owner and the resolver in
           the ENS (RegistrarDomain)", async function(){
66          let instance = getInstance('MultiSigWallet', accounts
               [0]);
67          const MultisigInstance = await instance
68              .deploy({ arguments: [multisigWalletSettings.owners
                   , multisigWalletSettings.required]}).send({from:
                    accounts[0]});
69
70          instance = getInstance('RegistrarDomain', accounts[0]);
71          const RegistrarDomainInstance = await instance
72              .deploy({ arguments: [registrarDomainSettings.root,
                    MultisigInstance._address]}).send({from:
                   accounts[0]});
73
74          let ENSaddress = await RegistrarDomainInstance.methods.
               getENS().call();
75
76          assert.notEqual(ENSaddress, null, "It does not have ENS
               ");
77
78          const ENSinstance = await new web3_1_0.eth.Contract(
               ENSABI, ENSaddress);
79          let domain = web3_1_0.utils.keccak256(".nounetwork");
80          let addr = await ENSinstance.methods.owner(domain).call
               ();
81
82          assert.equal(addr, RegistrarDomainInstance._address, "
               Owner in ENSRegistry has not been set correctly");
83      });
84
85      it("should have Resolver (RegistrarDomain)", async function
           (){
86          let instance = getInstance('MultiSigWallet', accounts
```

```
            [0]);
87          const MultisigInstance = await instance
88              .deploy({ arguments: [multisigWalletSettings.owners
                    , multisigWalletSettings.required]}).send({from:
                     accounts[0]});
89
90          instance = getInstance('RegistrarDomain', accounts[0]);
91          const RegistrarDomainInstance = await instance
92              .deploy({ arguments: [registrarDomainSettings.root,
                     MultisigInstance._address]}).send({from:
                    accounts[0]});
93
94          let resolverAddress = await RegistrarDomainInstance.
                methods.getResolver().call();
95
96          assert.notEqual(resolverAddress, null, "It does not
                have Resolver");
97      });
98
99      it("should set a new domain (RegistrarDomain)", async
            function(){
100         let instance = getInstance('MultiSigWallet', accounts
                [0]);
101         const MultisigInstance = await instance
102             .deploy({ arguments: [multisigWalletSettings.owners
                    , multisigWalletSettings.required]}).send({from:
                     accounts[0]});
103
104         instance = getInstance('RegistrarDomain', accounts[0]);
105         const RegistrarDomainInstance = await instance
106             .deploy({ arguments: [registrarDomainSettings.root,
                     MultisigInstance._address]}).send({from:
                    accounts[0]});
107
108         let ENSaddress = await RegistrarDomainInstance.methods.
                getENS().call();
109         const ENSinstance = await new web3_1_0.eth.Contract(
                ENSABI, ENSaddress);
110         let resolverAddress = await RegistrarDomainInstance.
                methods.getResolver().call();
111         const ResolverInstance = await new web3_1_0.eth.
                Contract(ResolverABI, resolverAddress);
112
113         let domain = web3_1_0.utils.keccak256('.upc');
114         await RegistrarDomainInstance.methods.createNodeDomain(
                domain).send({from: accounts[0]});
115         let root = await RegistrarDomainInstance.methods.
                getRoot().call();
116         domain = web3_1_0.utils.soliditySha3(root, domain);
117         let dom = await RegistrarDomainInstance.methods.
```

```
            getDomain().call();
118         assert.equal(domain, dom, "It does not have setted a
               new domain");
119
120         let registrarSubDomainAddr = await
               RegistrarDomainInstance.methods.
               getRegistrarSubDomain().call();
121
122         assert.notEqual(registrarSubDomainAddr, null, "It does
               not have setted a new domain");
123
124         let addr = await ENSinstance.methods.owner(domain).call
               ();
125
126         assert.equal(addr, registrarSubDomainAddr, "Owner in
               ENSRegistry has not been set correctly");
127
128         let resolver = await ENSinstance.methods.resolver(
               domain).call();
129
130         assert.equal(resolver, resolverAddress, "Resolver in
               ENSRegistry has not been set correctly");
131
132         addr = await ResolverInstance.methods.addr(domain).call
               ();
133
134         assert.equal(addr, registrarSubDomainAddr, "Owner in
               Resolver has not been set correctly");
135     });
136
137     it("should NOT set a new domain (RegistrarDomain) (sender
           is not an owner)", async function(){
138         let instance = getInstance('MultiSigWallet', accounts
               [0]);
139         const MultisigInstance = await instance
140             .deploy({ arguments: [multisigWalletSettings.owners
                   , multisigWalletSettings.required]}).send({from:
                    accounts[0]});
141
142         instance = getInstance('RegistrarDomain', accounts[0]);
143         const RegistrarDomainInstance = await instance
144             .deploy({ arguments: [registrarDomainSettings.root,
                    MultisigInstance._address]}).send({from:
                   accounts[0]});
145
146         await RegistrarDomainInstance.methods.multisigControl(
               true).send({from: accounts[0]});
147         let domain = web3_1_0.utils.keccak256('.upc');
148         await RegistrarDomainInstance.methods.createNodeDomain(
               domain).send({from: accounts[3]}).should.be.
```

```
            rejectedWith(ERROR_MSG);
149     });
150
151     it("should NOT set a new domain (RegistrarDomain) (domain
            already exists)", async function(){
152         let instance = getInstance('MultiSigWallet', accounts
                [0]);
153         const MultisigInstance = await instance
154             .deploy({ arguments: [multisigWalletSettings.owners
                    , multisigWalletSettings.required]}).send({from:
                     accounts[0]});
155
156         instance = getInstance('RegistrarDomain', accounts[0]);
157         const RegistrarDomainInstance = await instance
158             .deploy({ arguments: [registrarDomainSettings.root,
                     MultisigInstance._address]}).send({from:
                    accounts[0]});
159
160         let domain = web3_1_0.utils.keccak256('.upc');
161         await RegistrarDomainInstance.methods.multisigControl(
                true).send({from: accounts[0]});
162         await RegistrarDomainInstance.methods.createNodeDomain(
                domain).send({from: accounts[0]});
163         await RegistrarDomainInstance.methods.createNodeDomain(
                domain).send({from: accounts[1]}).should.be.
                rejectedWith(ERROR_MSG);
164     });
165
166     it("should NOT set a new domain (RegistrarDomain) (owner
            has already have a domain)", async function(){
167         let instance = getInstance('MultiSigWallet', accounts
                [0]);
168         const MultisigInstance = await instance
169             .deploy({ arguments: [multisigWalletSettings.owners
                    , multisigWalletSettings.required]}).send({from:
                     accounts[0]});
170
171         instance = getInstance('RegistrarDomain', accounts[0]);
172         const RegistrarDomainInstance = await instance
173             .deploy({ arguments: [registrarDomainSettings.root,
                     MultisigInstance._address]}).send({from:
                    accounts[0]});
174
175         await RegistrarDomainInstance.methods.multisigControl(
                true).send({from: accounts[0]});
176         await RegistrarDomainInstance.methods.setOnlyOneDomain(
                true).send({from: accounts[0]});
177         let domain = web3_1_0.utils.keccak256('.upc');
178         await RegistrarDomainInstance.methods.createNodeDomain(
                domain).send({from: accounts[0]});
```

```
179         domain = web3_1_0.utils.keccak256('.uab');
180         await RegistrarDomainInstance.methods.createNodeDomain(
               domain).send({from: accounts[0]}).should.be.
               rejectedWith(ERROR_MSG);
181     });
182
183     it("should have set a new subdomain (RegistrarSubDomain)",
           async function(){
184
185         let instance = getInstance('MultiSigWallet', accounts
               [0]);
186         const MultisigInstance = await instance
187             .deploy({ arguments: [multisigWalletSettings.owners
                   , multisigWalletSettings.required]}).send({from:
                    accounts[0]});
188
189         instance = getInstance('RegistrarDomain', accounts[0]);
190         const RegistrarDomainInstance = await instance
191             .deploy({ arguments: [registrarDomainSettings.root,
                    MultisigInstance._address]}).send({from:
                   accounts[0]});
192
193         let ENSaddress = await RegistrarDomainInstance.methods.
               ens().call();
194         const ENSinstance = await new web3_1_0.eth.Contract(
               ENSABI, ENSaddress);
195         let resolverAddress = await RegistrarDomainInstance.
               methods.resolver().call();
196         const ResolverInstance = await new web3_1_0.eth.
               Contract(ResolverABI, resolverAddress);
197
198         let domain = web3_1_0.utils.keccak256('.upc');
199         let root = await RegistrarDomainInstance.methods.
               getRoot().call();
200         await RegistrarDomainInstance.methods.createNodeDomain(
               domain).send({from: accounts[0]});
201
202         let registrarSubDomainAddr = await
               RegistrarDomainInstance.methods.
               getRegistrarSubDomain().call();
203         domain = web3_1_0.utils.soliditySha3(root, domain);
204         let owner = await ENSinstance.methods.owner(domain).
               call();
205         assert.equal(owner, registrarSubDomainAddr, "Owner is
               not the subdomain registrar");
206         owner = await ResolverInstance.methods.addr(domain).
               call();
207         assert.equal(owner, registrarSubDomainAddr, "Address of
                the domain has not been set correctly in the
               Resolver")
```

```
208
209         let resolver = await ENSinstance.methods.resolver(
                domain).call();
210         assert.equal(resolver, resolverAddress, "Resolver is
                not correct");
211
212         const RegistrarSubDomainInstance = await new web3_1_0.
                eth.Contract(RegistrarSubDomainABI,
                registrarSubDomainAddr);
213         let subdomain = web3_1_0.utils.keccak256('alberto');
214
215         //ERROR setsubdomainowner()
216         await RegistrarSubDomainInstance.methods.
                createUserSubDomain(subdomain).send({from: accounts
                [2], gas: 126000});
217         subdomain = web3_1_0.utils.solidity.Sha3(domain,
                subdomain);
218         owner = await ENSinstance.methods.owner(subdomain).call
                ();
219         let addr = await ResolverInstance.methods.addr(
                subdomain).call();
220         resolver = await ENSinstance.methods.resolver(subdomain
                ).call();
221
222         assert.equal(resolver, resolverAddress, "Resolver of
                the subdomain has not been set correctly");
223         assert.equal(owner, web3_1_0.utils.toChecksumAddress(
                accounts[2]), "Owner of subdomain is not correct");
224         assert.equal(addr, web3_1_0.utils.toChecksumAddress(
                accounts[2]), "Owner in Resolver has not been set
                correctly");
225     });
226 });
227
228
229 contract('ENSRegistry', function(accounts){
230
231     it("should have a root owner", async function(){
232         const instance = getInstance('ENSRegistry', accounts
                [0]);
233         const ENSinstance = await instance
234             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
235         ensSettings.address = ENSinstance._address;
236         let node = await ENSinstance.methods.owner(ensSettings.
                root).call();
237
238         assert.equal(node, web3_1_0.utils.toChecksumAddress(
                ensSettings.owner), "It does not have owner");
239     });
```

```
240
241     it("should set a new domain owner", async function(){
242         const instance = getInstance('ENSRegistry', accounts
                [0]);
243         const ENSinstance = await instance
244             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
245
246         let root = web3_1_0.utils.keccak256('.nounetwork');
247         let domain = web3_1_0.utils.keccak256('.upc');
248
249         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[2]).send({from: accounts[0]});
250         domain = web3_1_0.utils.solioditySha3(root, domain);
251         let owner = await ENSinstance.methods.owner(domain).
                call();
252         assert.equal(owner, web3_1_0.utils.toChecksumAddress(
                accounts[2]), "It does not set a new domain owner");
253     });
254
255     it("should NOT set a new domain owner (sender is not the
            owner of the root)", async function(){
256         const instance = getInstance('ENSRegistry', accounts
                [0]);
257         const ENSinstance = await instance
258             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
259
260         let root = web3_1_0.utils.keccak256('.nounetwork');
261         let domain = web3_1_0.utils.keccak256('.upc');
262
263         //Sender is not the owner of the root domain, the owner
                it's accounts[0]
264         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[3]).send({from: accounts[3]}).should.be.
                rejectedWith(ERROR_MSG);
265     });
266
267     it("should NOT set a new domain owner (domain already
            exists)", async function(){
268         const instance = getInstance('ENSRegistry', accounts
                [0]);
269         const ENSinstance = await instance
270             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
271
272         let root = web3_1_0.utils.keccak256('.nounetwork');
273         let domain = web3_1_0.utils.keccak256('.upc');
274
275         await ENSinstance.methods.setDomainOwner(root, domain,
```

```
            accounts[3]).send({from: accounts[0]});
276
277         //The domain ".upc" already exists
278         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[3]).send({from: accounts[0]}).should.be.
               rejectedWith(ERROR_MSG);
279     });
280
281     it("should NOT set a new domain owner (owner has already
           have a domain)", async function(){
282         const instance = getInstance('ENSRegistry', accounts
               [0]);
283         const ENSinstance = await instance
284             .deploy({ arguments: [ensSettings.owner,
                   ensSettings.root]}).send({from: accounts[0]});
285
286         let root = web3_1_0.utils.keccak256('.nounetwork');
287         let domain = web3_1_0.utils.keccak256('.upc');
288
289         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[3]).send({from: accounts[0]});
290
291         domain = web3_1_0.utils.keccak256('.uab');
292         //The owner accounts[3] has already have the domain ".
               upc.nounetwork"
293         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[3]).send({from: accounts[0]}).should.be.
               rejectedWith(ERROR_MSG);
294     });
295
296     it("should set a new sub domain owner", async function(){
297         const instance = getInstance('ENSRegistry', accounts
               [0]);
298         const ENSinstance = await instance
299             .deploy({ arguments: [ensSettings.owner,
                   ensSettings.root]}).send({from: accounts[0]});
300
301         let root = web3_1_0.utils.keccak256('.nounetwork');
302         let domain = web3_1_0.utils.keccak256('.upc');
303         let subdomain = web3_1_0.utils.keccak256('alberto');
304
305         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[0]).send({from: accounts[0]});
306         domain = web3_1_0.utils.soliditySha3(root, domain);
307         await ENSinstance.methods.setSubDomainOwner(domain,
               subdomain, accounts[3]).send({from: accounts[0]});
308
309         subdomain = web3_1_0.utils.soliditySha3(domain,
               web3_1_0.utils.keccak256('alberto'));
310         let owner = await ENSinstance.methods.owner(subdomain).
```

```
            call();
311
312         assert.equal(owner, web3_1_0.utils.toChecksumAddress(
               accounts[3]), "It does not create a new sub domain")
               ;
313     });
314
315     it("should NOT set a new sub domain owner (sender is not
           the owner of the domain)", async function(){
316         const instance = getInstance('ENSRegistry', accounts
               [0]);
317         const ENSinstance = await instance
318             .deploy({ arguments: [ensSettings.owner,
                   ensSettings.root]}).send({from: accounts[0]});
319
320         let root = web3_1_0.utils.keccak256('.nounetwork');
321         let domain = web3_1_0.utils.keccak256('.upc');
322         let subdomain = web3_1_0.utils.keccak256('alberto');
323
324         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[0]).send({from: accounts[0]});
325         domain = web3_1_0.utils.soliditySha3(root, domain);
326         await ENSinstance.methods.setSubDomainOwner(domain,
               subdomain, accounts[3]).send({from: accounts[2]}).
               should.be.rejectedWith(ERROR_MSG);
327     });
328
329     it("should NOT set a new sub domain owner (subdomain
           already exists)", async function(){
330         const instance = getInstance('ENSRegistry', accounts
               [0]);
331         const ENSinstance = await instance
332             .deploy({ arguments: [ensSettings.owner,
                   ensSettings.root]}).send({from: accounts[0]});
333
334         let root = web3_1_0.utils.keccak256('.nounetwork');
335         let domain = web3_1_0.utils.keccak256('.upc');
336         let subdomain = web3_1_0.utils.keccak256('alberto');
337
338         await ENSinstance.methods.setDomainOwner(root, domain,
               accounts[0]).send({from: accounts[0]});
339         domain = web3_1_0.utils.soliditySha3(root, domain);
340         await ENSinstance.methods.setSubDomainOwner(domain,
               subdomain, accounts[2]).send({from: accounts[0]});
341         await ENSinstance.methods.setSubDomainOwner(domain,
               subdomain, accounts[3]).send({from: accounts[0]}).
               should.be.rejectedWith(ERROR_MSG);
342     });
343
344     it("should NOT set a new sub domain owner (owner has
```

```
        already have a subdomain)", async function(){
345         const instance = getInstance('ENSRegistry', accounts
                [0]);
346         const ENSinstance = await instance
347             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
348
349         let root = web3_1_0.utils.keccak256('.nounetwork');
350         let domain = web3_1_0.utils.keccak256('.upc');
351         let subdomain = web3_1_0.utils.keccak256('alberto');
352
353         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[0]).send({from: accounts[0]});
354         domain = web3_1_0.utils.solidity.Sha3(root, domain);
355         await ENSinstance.methods.setSubDomainOwner(domain,
                subdomain, accounts[2]).send({from: accounts[0]});
356         subdomain = web3_1_0.utils.keccak256('jose');
357         await ENSinstance.methods.setSubDomainOwner(domain,
                subdomain, accounts[2]).send({from: accounts[0]}).
                should.be.rejectedWith(ERROR_MSG);
358     });
359
360     it("should set a new resolver", async function(){
361         const instance = getInstance('ENSRegistry', accounts
                [0]);
362         const ENSinstance = await instance
363             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
364
365         let root = web3_1_0.utils.keccak256('.nounetwork');
366         let domain = web3_1_0.utils.keccak256('.upc');
367
368         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[0]).send({from: accounts[0]});
369         await ENSinstance.methods.setResolver(root, domain,
                accounts[3]).send({from: accounts[0]});
370         domain = web3_1_0.utils.solidity.Sha3(root, domain);
371         let resolver = await ENSinstance.methods.resolver(
                domain).call();
372         assert.equal(resolver, web3_1_0.utils.toChecksumAddress
                (accounts[3]));
373     });
374
375     it("should NOT set a new resolver (sender is not the owner
        of the domain)", async function(){
376         const instance = getInstance('ENSRegistry', accounts
                [0]);
377         const ENSinstance = await instance
378             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
```

```
379
380         let root = web3_1_0.utils.keccak256('.nounetwork');
381         let domain = web3_1_0.utils.keccak256('.upc');
382
383         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[0]).send({from: accounts[0]});
384         await ENSinstance.methods.setResolver(root, domain,
                accounts[3]).send({from: accounts[1]}).should.be.
                rejectedWith(ERROR_MSG);
385     });
386
387     it("should change the domain owner", async function(){
388         const instance = getInstance('ENSRegistry', accounts
                [0]);
389         const ENSinstance = await instance
390             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
391
392         let root = web3_1_0.utils.keccak256('.nounetwork');
393         let domain = web3_1_0.utils.keccak256('.upc');
394         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[0]).send({from: accounts[0]});
395         domain = web3_1_0.utils.soliditySha3(root, domain);
396         await ENSinstance.methods.setOwner(domain, accounts[3])
                .send({from: accounts[0]});
397         let owner = await ENSinstance.methods.owner(domain).
                call();
398         assert.equal(owner, web3_1_0.utils.toChecksumAddress(
                accounts[3]));
399     });
400
401     it("should NOT change the domain owner (sender is not the
            owner of the domain)", async function(){
402         const instance = getInstance('ENSRegistry', accounts
                [0]);
403         const ENSinstance = await instance
404             .deploy({ arguments: [ensSettings.owner,
                    ensSettings.root]}).send({from: accounts[0]});
405         let root = web3_1_0.utils.keccak256('.nounetwork');
406         let domain = web3_1_0.utils.keccak256('.upc');
407         await ENSinstance.methods.setDomainOwner(root, domain,
                accounts[0]).send({from: accounts[0]});
408         domain = web3_1_0.utils.soliditySha3(root, domain);
409         await ENSinstance.methods.setOwner(domain, accounts[3])
                .send({from: accounts[1]}).should.be.rejectedWith(
                ERROR_MSG);
410     });
411
412 });
413
```

```
414 contract('Resolver', function(accounts){
415     it("should have ENS", async function() {
416         const instance = getInstance('Resolver', accounts[0]);
417         const ResolverInstance = await instance
418             .deploy({ arguments: [ensSettings.address]}).send({
                    from: accounts[0]});
419         let ens = await ResolverInstance.methods.getENS().call
                ();
420 
421         assert.equal(ens, ensSettings.address, "ENS is not
                setted correctly");
422     });
423 
424     it("should support the address interface", async function()
            {
425         const instance = getInstance('Resolver', accounts[0]);
426         const ResolverInstance = await instance
427             .deploy({ arguments: [ensSettings.address]}).send({
                    from: accounts[0]});
428         let interface = await ResolverInstance.methods.
                supportsInterface("0x3b3b57de").call();
429 
430         assert.equal(interface, true, "Resolver does not
                support address interface");
431     });
432 
433     it("should NOT map a domain to an address (sender is not
            the owner of the domain)", async function() {
434         const instance = getInstance('Resolver', accounts[0]);
435         const ResolverInstance = await instance
436             .deploy({ arguments: [ensSettings.address]}).send({
                    from: accounts[0]});
437         let root = web3_1_0.utils.keccak256(".nounetwork");
438         let domain = web3_1_0.utils.keccak256(".upc");
439         await ResolverInstance.methods.setAddr(root, domain,
                accounts[3]).send({from: accounts[0]}).should.be.
                rejectedWith(ERROR_MSG);
440     });
441 
442     it("should NOT map a domain to an address (address is null)
            ", async function() {
443         const instance = getInstance('Resolver', accounts[0]);
444         const ResolverInstance = await instance
445             .deploy({ arguments: [ensSettings.address]}).send({
                    from: accounts[0]});
446         let root = web3_1_0.utils.keccak256(".nounetwork");
447         let domain = web3_1_0.utils.keccak256(".upc");
448         await ResolverInstance.methods.setAddr(root, domain, "0
                x0000000000000000000000000000000000000000").send({
                from: accounts[0]}).should.be.rejectedWith(ERROR_MSG
```

```
                );
449         });
450 });
```

Bibliography

[1] G2Crowd. https://www.g2crowd.com/categories/blockchain. Last access 2018.

[2] Lisk Documentation. https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/what-is-a-peer-to-peer-network. Last access 2018.

[3] Ethereum - BlockGeecks. https://blockgeeks.com/guides/ethereum/. Last access 2018.

[4] Ethereum Name Service - Read the Docs. https://docs.ens.domains/en/latest/. Last access 2018.

[5] Alastria - National Blockchain Ecosystem. https://alastria.io. Last access 2018.

[6] Bob Marvin. Blockchain: The Invisible Technology That's Changing the World. https://au.pcmag.com/features/46389/blockchain-the-invisible-technology-thats-changing-the-world, 30 August 2017. Last access 2018.

[7] Ethereum Project. https://www.ethereum.org/. Last access 2018.

[8] Hiperledger Fabric - Hyperledger. https://www.hyperledger.org/projects/fabric. Last access 2018.

[9] El Comuns de la Xarxa Oberta, Lliure i Neutral. https://guifi.net/ComunsXOLN, December 26, 2009. Last access 2018.

[10] Quorum - J.P. Morgan. https://www.jpmorgan.com/global/Quorum. Last access 2018.

[11] Hiperledger Fabric - Read the Docs. https://hyperledger-fabric.readthedocs.io/en/release-1.2/index.html. Last access 2018.

[12] Alicia Naumoff. Why Blockchain Needs 'Proof of Authority' Instead of 'Proof of Stake'. https://cointelegraph.com/news/why-blockchain-needs-proof-of-authority-instead-of-proof-of-stake, April 26, 2017. Last access 2018.

[13] Collin Cusce. Using puppeth To Manually Create An Ethereum Proof Of Authority Network. https://medium.com/@collin.cusce/using-puppeth-to-manually-create-an-ethereum-proof-of-authority-clique-network-on-aws-ae0d7c906cce, June 23, 2018. Last access 2018.

[14] Geth Command Line Options - GitHub. https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options, June 19, 2018. Last access 2018.

[15] James Ray. Network Status Monitoring. https://github.com/ethereum/wiki/wiki/Network-Status#network-status-monitoring, August 22, 2018. Last access 2018.

[16] Ethereum Name Service. https://ens.domains/. Last access 2018.

[17] Nick Johnson. State of the ENS: Week 8. https://medium.com/the-ethereum-name-service/state-of-the-ens-week-8-86cd6ee23e66, July 4, 2017. Last access 2018.

[18] Stefan George. Gnosis Multisig. https://github.com/gnosis/MultiSigWallet/blob/master/contracts/MultiSigWallet.sol, 16 Dec 2017. Last access 2018.